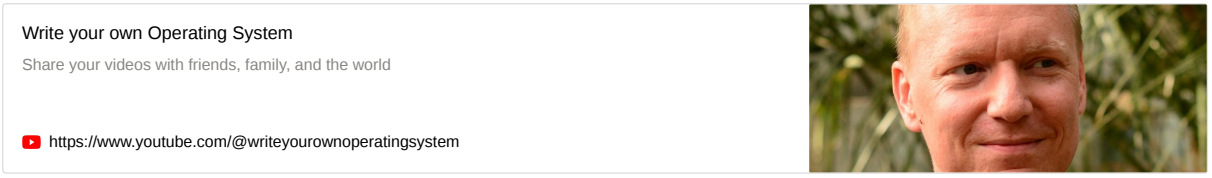
Faire son propre OS (cpp) - le PDF

Credit : Je me sers en grande partie de cette chaine pour réaliser mon OS. Le pdf contient les informations utiles contenues dans ses vidéos.



sites utiles pour la création d’un OS :

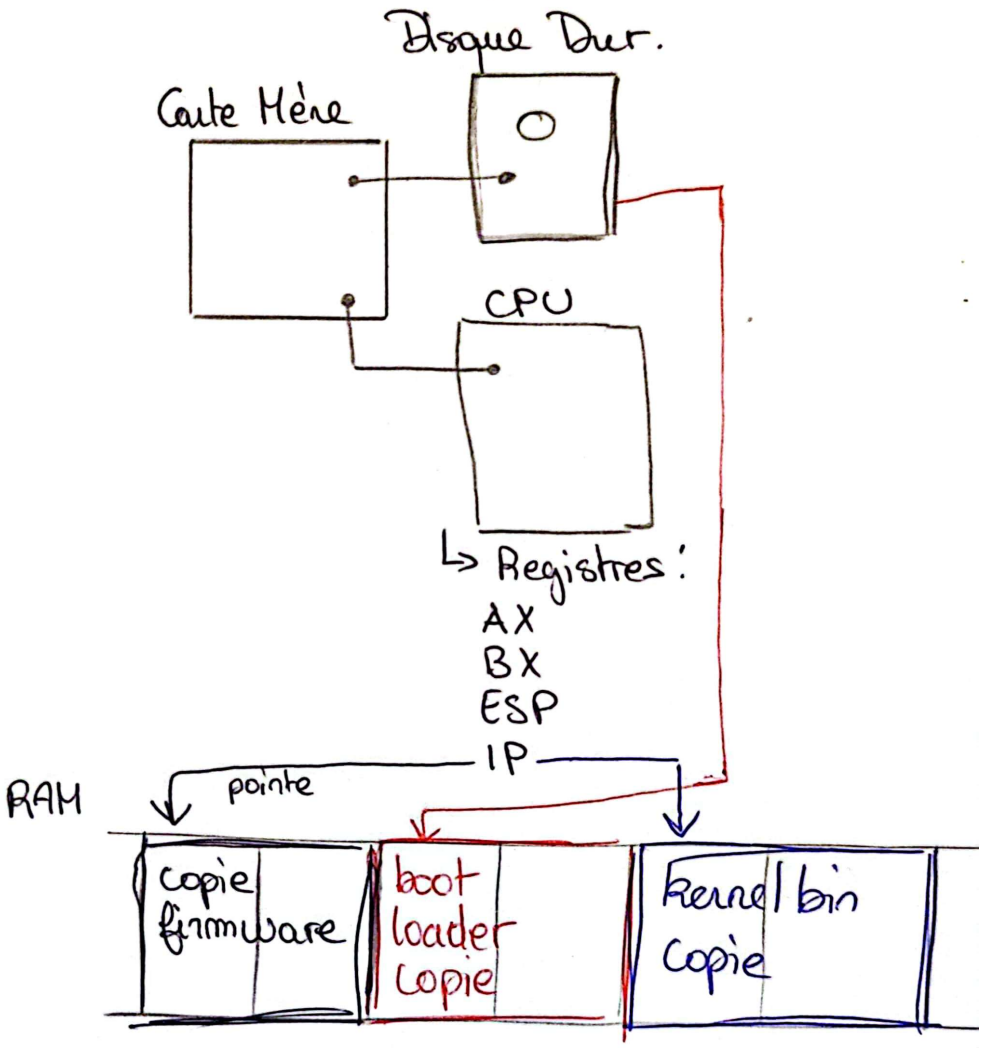
OSDev Wiki

Lowlevel.eu

Dépendances :

```
sudo apt-get install g++
                     binutils
                     libc6-dev-i386
                     xorriso
```

Comprendre le démarrage d’un ordinateur.



- Au démarrage, la carte mère va copier les données du BIOS (assembleur) dans la RAM, que l'on appelle le firmware (microcode, microprogramme). Il indique au CPU de pointer le registre IP (Instruction Pointer), ce qui permet au CPU de lire et exécuter les informations contenues dans la RAM. (flèche noire)
- Le firmware indique au CPU de lire le contenu du disque dur, qui sera chargé dans la RAM (boot loader). Ensuite, l'adresse du registre IP passe à celle du boot loader en RAM pour exécuter ce dernier. (flèche rouge)
- Le bootloader, complexe, gère les systèmes de fichiers ainsi que les partitions. Cela permet d'accéder au chemin `/boot/grub/grub.cfg`, qui contient le menu des entrées menant aux OS (en gros il permet de situer dans le DD les OS) (c'est la partie qui affiche les OS accessible lors du démarrage)
- Une fois choisie, le bootloader effectue une copie du noyau (ici kernelbin) dans la RAM et fait pointer le registre IP vers cette copie, ce qui permet à l'ordinateur de charger le système d'exploitation.

Problème 1 :

> Le boot loader n'initialise pas le registre ESP (Stack Pointer). Cependant CPP nécessite d'avoir ce registre initialisé pour fonctionner correctement.

→ créer deux fichiers, `loader.s` en assembleur pour definir ESP, et sauter vers le deuxieme fichier CPP `kernel.cpp` qui contient l'OS

`loader.s` est compilé par GNU AS et donne `loader.o`, `kernel.cpp` qui contient l'OS.

est compilé par g++ et donne `kernel.o`

Problème 2 :

Les deux objets codés dans les langages différents doivent être combiné.

→ utilisation du programme LD. qui est un "linker" et donne un fichier kernel bin

nb : Le CPU par défaut au démarrage est compatible 32 bit, donc nous allons faire un noyau en 32 bit.

Problème 3 :

Le stack pointeur (négatif) peut pointer vers des emplacement mémoire de la RAM déjà utilisé et overwrite des informations, ce que l'on ne veut pas

→ il suffit d'allouer de la place afin de ne pas écrire (2mb suffisent)

Problème 4 :

Le boot loader ne va pas reconnaitre ce que l'on a fait comme un kernel, car il ne contient pas de metric number,

→ il suffit de le rajouter

IL EST : `0x1badb002`

La size de la RAM, le bootloader crée une structure et store a pointer vers cette structure dans le registre AX, et le metric number dans le registre BX

Problème 5 :

Comme nous codons en CPP, en dehors d'un OS.

```
#include <stdio.h>
```

N'existe pas. Ces bibliothèques ne sont pas contenues dans notre OS.

donc pas de dynamic linking, pas de memory management, pas de malloc etc…

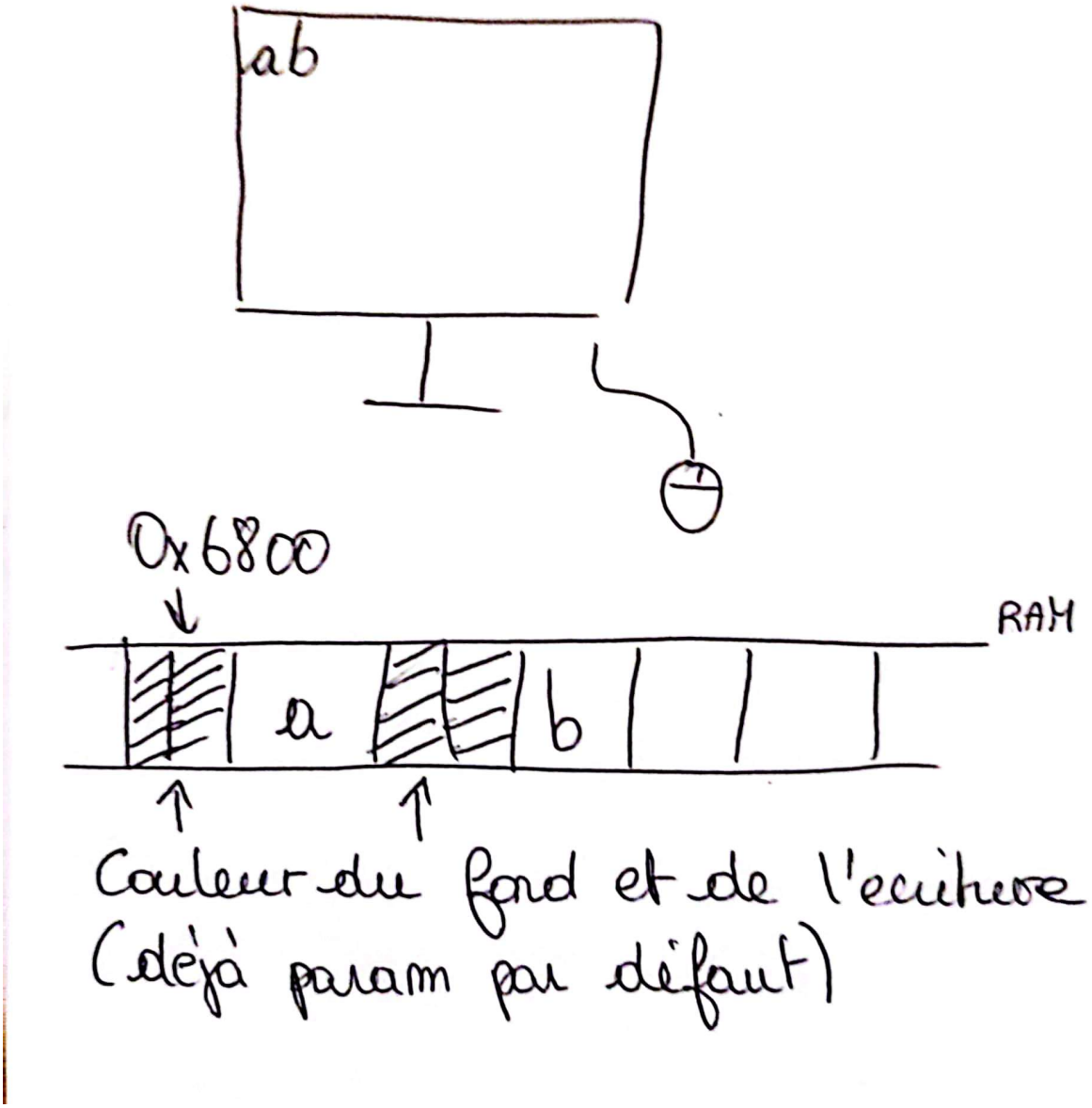
Solution : do it yourself :)

Nous devons donc écrire notre propre printf par exemple.

Affichage sur écran.

Lorsque l'on écrit dans la ram à l'emplacement : `0xb8000`

Ce que l'on écrit sur cette ligne sera affiché par la carte graphique.



Une partie de la mémoire est réservé pour la couleur et le fond de l'écriture, qui est par défaut fond noir et police blanche.

Ajouter l'os dans grub multiboot.

Au moment où j'écris ce texte. Il faut avoir grub sur son ordinateur, je ne sais pas si je vais coder un multiboot, car ma VM kali le contient déjà et est simple à mettre en place.

make …

`sudo vim /boot/grub/grub.cfg`

```
### BEGIN MYKERNEL ###

menuentry 'Le nom de votre OS qui s affichera lors du démarrage.'{
  multiboot /boot/mykernel.bin
  boot
}

### END MYKERNEL ###
```

```
:w!
:q
```

Ajouter l’OS sur VirtualBox.

Dans virtualbox quand on crée une machine → Type: Other, Version: Other/Unknown

En ajoutant l’iso on tourne l’OS sur VM.

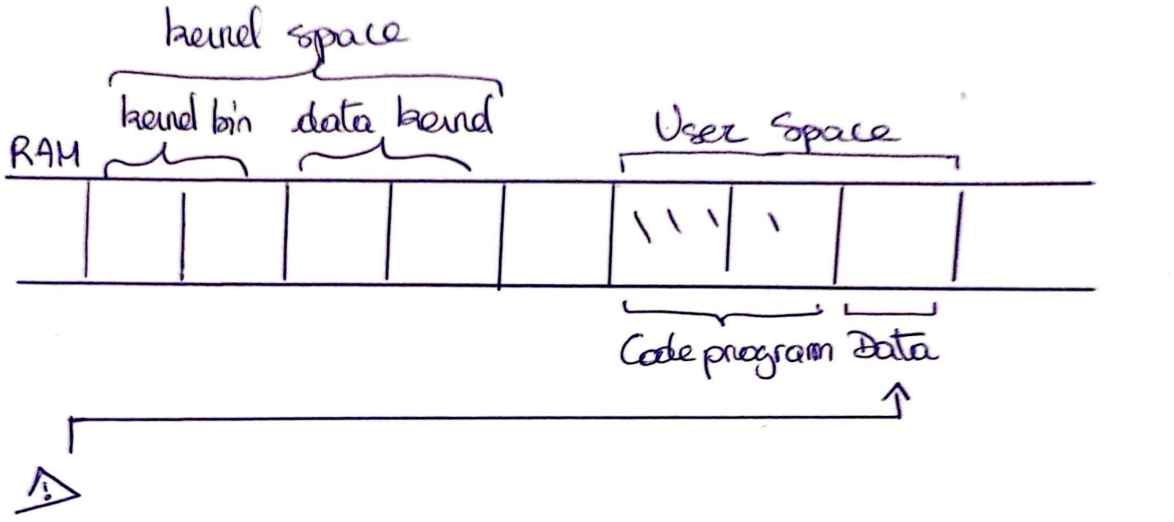
NB: la commande make run permet de lancer la VM

Hardware communication.

La communication avec le hardware est très précise, nous devons savoir précisément sur combien de bit la data est codé, et où elle se situe.

Dans l’un de nos fichier. nous avions déclaré un unsigned int en 4 bit, dans notre programme il faut le déclarer de telle sorte. C’est le rôle du fichier `types.h` qui définie ces types. que nous utiliserons tout le temps par la suite.

Memory segment.



/!\ Une attaque qui existait, était de charger du code binaire dans la partie de mémoire réservé a la data. Que l’on appelle `Data Segment` (Aujourd’hui il n’est plus possible de pouvoir jump dans cette partie), la partie qui contient le code binaire “Code program” est appelé `Code Segment`.

Logique d’entrées du clavier.

Lorsque l'on clique sur une touche du clavier. Le voltage traverse le câble et le processeur lit cette information.

→ Cela peut mener a une interruption qui risque de faire sauter l'ordinateur à une adresse mémoire différente.

Il faut ainsi réaliser un `Interrupt descriptor table` qui contient les informations pour switch ou non la memory segment.

Afin de gérer les segments, nous devons réaliser une `Global Descriptor Table (GDT)`

Global Descriptor Table et difficultés

La table globale de description est une table qui contient les informations suivantes

- Point de départ du segment
- Taille du segment
- Des flags tq: s'il s'agit d'un Code Segment ou non, si le fichier est un exécutable ou non, quels droits à besoin le fichier pour faire un saut à son adresse..

Elle est codé sous 8 Octets (Bytes) et la difficulté repose sur la manière dont ces 8 octets sont écris. En effet le GDT doit être retrocompatible (backward compatible). c'est à dire qu'il doit fonctionner avec des versions antérieurs tq les processeurs 386 avec des adresses de 16 bits,ui donne une écriture des bits très compliqué.

Ce qui donnerait une représentation de la sorte :

pointer	limite (4bit) | flag (4bit)	Droits d'accès	pointer	pointer	pointer	limite	limite

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Avec : pointer, limite, flag, droits d'accès.

on a donc des informations dont les bits ne sont pas succincts. qui rend l'opération plus lourde et complexe. en effet pour lire les informations du pointeur. il faut lire toutes les informations en rouge. et donc sauter les bits intermédiaires. Une opération à coder à la main.

les constructeurs dans le linker pour qu'ils soient dans le binaire, mais ils ne sont jamais appellés

c'est un probleme pour les objets globaux, et statics, les classes, et pourles objets composites. structures ?

Il faut utiliser les pointeurs qui sont des types primitifs 😿. pour eviter ce genre de probleme